

If you've written something like `x = x + 1`, you've already used a JavaScript operator. They show up in almost every line of code—whether you're doing a quick calculation, making a comparison, or updating a value.

What many beginners don't realize is that JavaScript keeps adding new kinds of operators over time. Some of them, like `?.` (**optional chaining**) or `??` (**nullish coalescing**), are designed to help you write cleaner, more reliable code—especially when you're working with data that might be missing or undefined.

You don't need to learn those right away. What's more important is building a solid foundation with the basics—things like math, assignments, and comparisons. Once you feel comfortable with those, the newer features will be much easier to pick up when you need them.

Arithmetic Operators: doing math

These are the first operators most people learn—and for good reason. They let you do simple math in your code:

```
let sum = 5 + 3; // 8
let difference = 10 - 4; // 6
let product = 6 * 7; // 42
let quotient = 8 / 2; // 4
let remainder = 10 % 3; // 1
let power = 2 ** 3; // 8
```

☐ ☐ ☐ ☐

Arithmetic operators are essential for everything from form input to game mechanics.

The + Operator: numbers or strings?

The `+` operator is a bit of a shape-shifter in JavaScript. Use it with numbers, and it adds. Use it with strings, and it joins them together.

```
5 + 5 // 10
"Hello" + "!" // "Hello!"
```

☐ ☐ ☐ ☐

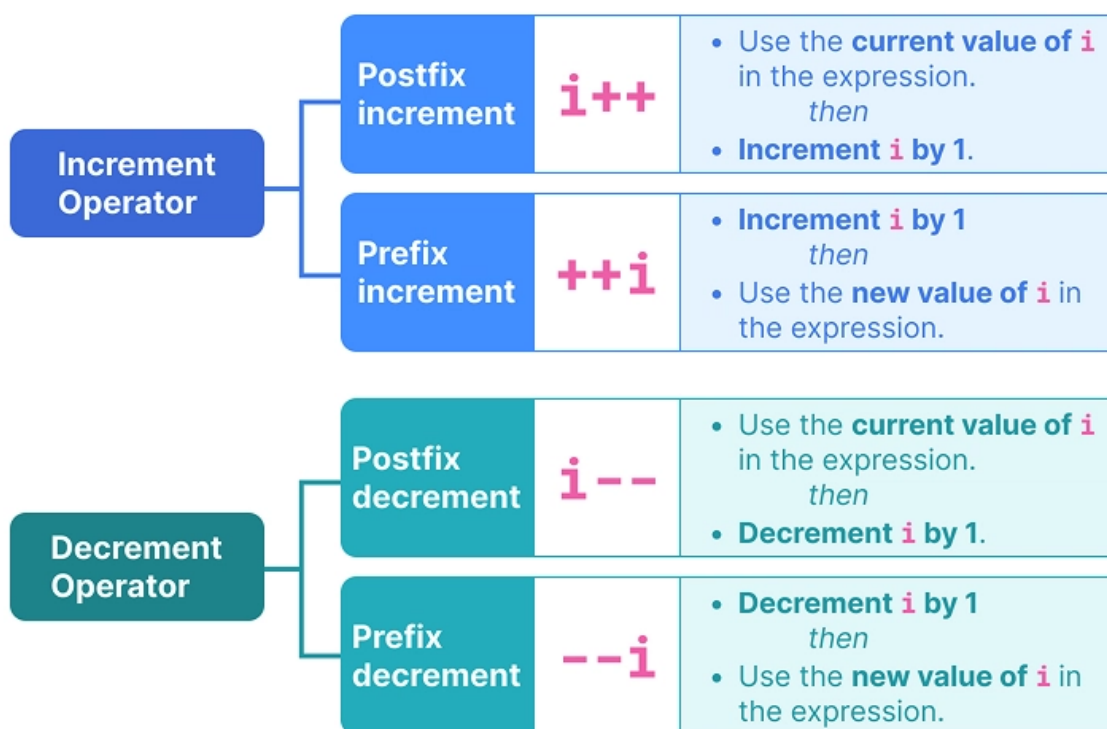
But combine a number and a string?

```
5 + "5" // "55" (string)
```

☐ ☐ ☐ ☐

JavaScript treats the number like a string and sticks them together. It's called type coercion—and it's a good reason to double-check your variable types when using `+`.

`++` and `--`: shortcuts that add (or subtract) 1



Want to increase a value by one? Use `++`. Decrease it? Use `--`. You'll see these a lot in loops.

```
let count = 1;
count++; // count is now 2
```

⌘ ⇧ ⇧ ⇧

There's a twist, though. These operators come in **prefix** and **postfix** forms:

```
let a = 5;
let b = a++; // b = 5, a = 6
let x = 5;
let y = ++x; // x = 6, y = 6
```

⌘ ⇧ ⇧ ⇧

Postfix returns the value before the change; prefix returns it after. This becomes important in more complex expressions.

Assignment Operators: more than just '='

At its core, `=` assigns a value. But JavaScript also lets you combine assignment with operations.

```
let score = 10;
score += 5; // Same as: score = score + 5
score *= 2; // Multiplies score by 2
```





These shortcuts make your code tighter—and easier to follow once you know what to look for.

Comparison Operators: making decisions

Comparison Operators

true ?


false ?

 > 

Compare a value in the left with a value in the right

when a = 1, b = "1"

==	equal to	a == b → true
===	equal value and data type	a === b → false
!=	not equal	a != b → false
!==	not equal value or data type	a !== b → true
>	greater than	a > b → false
<	less than	a < b → false
>=	greater than or equal to	a >= b → true
<=	less than or equal to	a <= b → true



These operators return *true* or *false*. You'll use them in every *if* statement and loop condition.

```
5 === 5 // true
5 !== "5" // true
10 > 5 // true
```



Use `===` and `!==` over `==` and `!=` to avoid unwanted type conversions. It's a small change that can save a lot of debugging.

Conditional (Ternary) Operators: one line, two outcomes

This one's a favorite for quick decisions:


```
let age = 20;
let status = age >= 18 ? "Adult" : "Minor";
```



It reads: “If age is 18 or more, return ‘Adult’; otherwise, return ‘Minor’.” Just don’t over-nest them. Simplicity wins.

Logical Operators: combining conditions

Logical Operators					JS
&& (AND)	true	&&	true	→	true
	true	&&	false	→	false
	false	&&	false	→	false
 (OR)	true		true	→	true
	true		false	→	true
	false		false	→	false
! (NOT)		!	true	→	false
		!	false	→	true



When one condition isn’t enough, logical operators help:

- && (AND): Both must be true
- || (OR): At least one must be true
- ! (NOT): Reverses the condition

```
let isMember = true;  
let hasAccess = isMember && age >= 18;
```

□ □ □ □

They’re essential when your code needs to respond to multiple factors at once.

Logical Assignment: a cleaner way to check and set

JavaScript now lets you combine logical checks and assignments in one line:

```
user.name ||= "Guest"; // Set if falsy
settings.apiKey ??= "none"; // Set if null or undefined
isAdmin &&= true; // Set only if already truthy
```

🔗🔗🔗

It's a modern way to reduce code clutter while still being clear about your intent.

Nullish Coalescing (??): a smarter default

This operator provides a fallback **only if** the first value is or *undefined*.

```
let value = input ?? "default";
```

🔗🔗🔗

It won't trigger on empty strings or 0—which makes it more precise than || in many cases.

Optional Chaining (?.): avoid errors on missing data

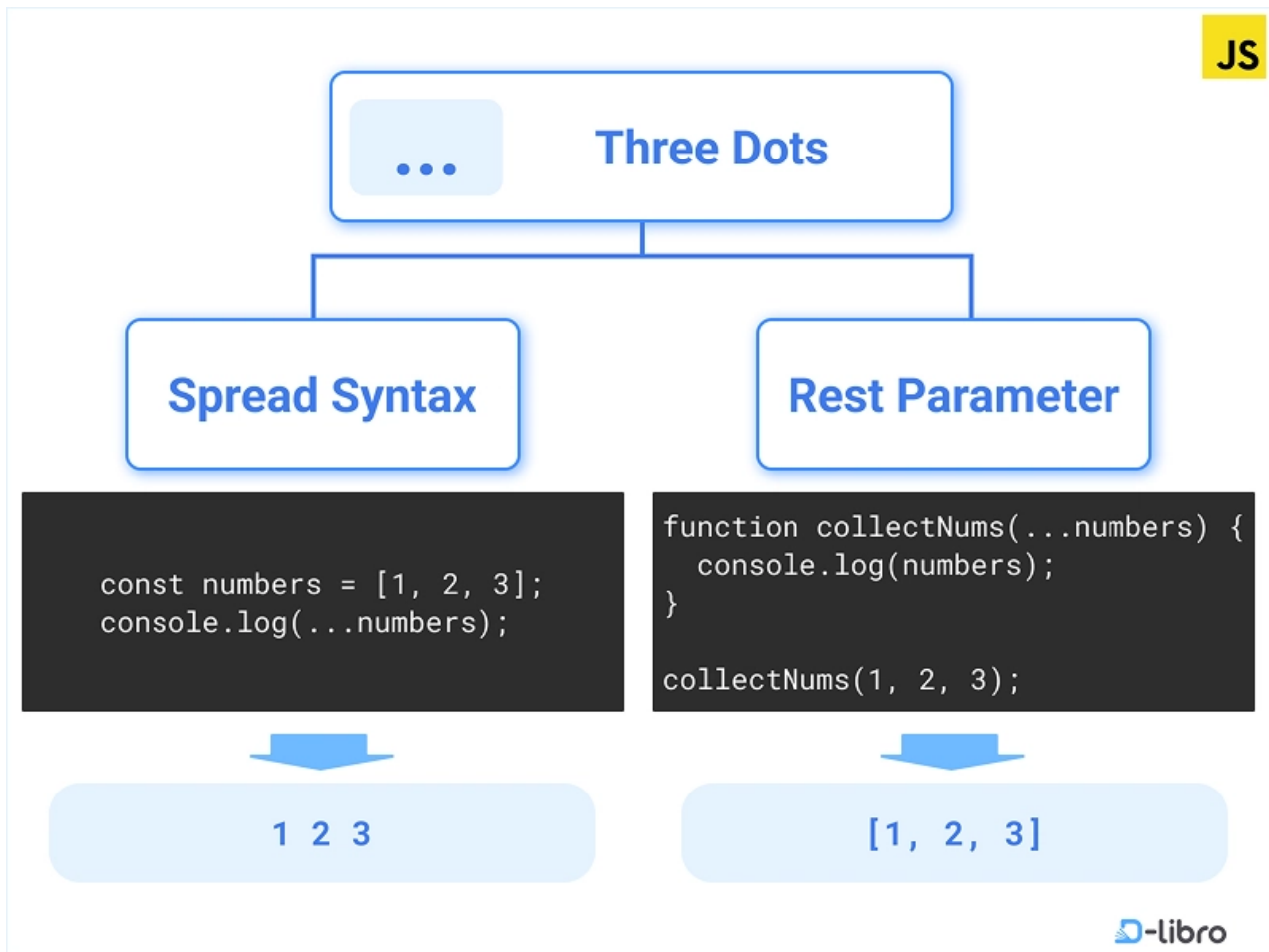
Need to access a property deep in an object, but not sure if the middle parts exist?

```
let name = user?.profile?.firstName;
```

🔗🔗🔗

If any part is undefined, it won't throw an error—it just returns undefined. It's perfect for working with optional data like API responses.

Spread and Rest: one syntax, two uses



The three dots (...) serve two purposes depending on context:

- **Spread:** Expands arrays or objects
- **Rest Parameter:** Collects remaining values

```
// Spread
let copy = [...original];
// Rest
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b);
}
```

☐ ☐ ☐ ☐

This syntax is everywhere in modern JavaScript, especially in frameworks like React.

Learn JavaScript Coding with AI:
Revolutionize Your Learning in This
Beginner's JavaScript Book

